Assignment- Design an End-to-End AI voice Assistant Pipeline

Objective

The objective of this solution is to build an end-to-end AI-powered voice assistant pipeline. This pipeline takes an audio input in MP3 format, applies voice activity detection (VAD) to isolate speech, transcribes the speech to text using a Whisper model, generates a response using a microsoft/phi-2 model, and converts the generated text back to speech using Edge-TTS. The solution also allows for user customization of voice type, speech rate, and pitch.

Choice of Models and Libraries

1. **Whisper (Speech-to-Text)**
    - **Library**: whisper
    - **Model**: Whisper model (base)
    - **Purpose**: Convert spoken language in the audio input to text. Whisper is chosen for its state-of-the-art accuracy and ability to handle diverse accents and background noises.
2. **Phi-2(Text Generation)**
    - **Library**: transformers
    - **Model**: Phi-2(loaded via AutoModelForCausalLM)
    - **Purpose**: Generate text responses with a focus on fast inference and low latency, making it ideal for applications where quick response times are crucial.
3. **Edge-TTS (Text-to-Speech)**
    - **Library**: edge_tts
    - **Purpose**: Convert the generated text back into speech. Edge-TTS is used for its high-quality, neural network-based voice synthesis, with customizable voice parameters.
4. **WebRTC VAD (Voice Activity Detection)**
    - **Library**: webrtcvad

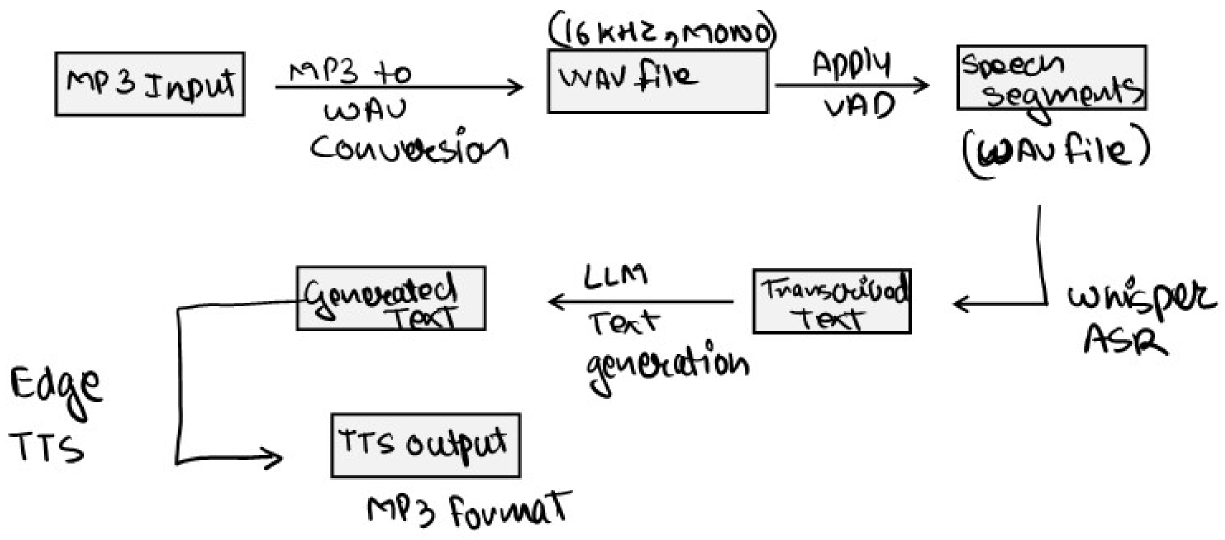
- **Purpose**: Isolate speech segments from silence and noise in the audio. WebRTC VAD is known for its robustness in detecting speech in various acoustic environments.

5. **Pydub (Audio Processing)**
   - **Library**: pydub
   - **Purpose**: Convert MP3 files to WAV format and adjust audio parameters such as sample rate and channels. Pydub provides a simple interface for audio file manipulation.

Parameters Used

1. **VAD Aggressiveness**: Set to 3 for higher sensitivity to speech, ensuring that even soft or faint speech is detected.
2. **Sample Rate**: 16kHz (16000 samples per second), as required by the WebRTC VAD and recommended for ASR models.
3. **Channels**: Mono (single channel), to simplify processing and ensure compatibility with VAD.
4. **Text Generation Constraints**: Maximum of 200 tokens and a restriction to 2 sentences for concise response

Architecture Diagram

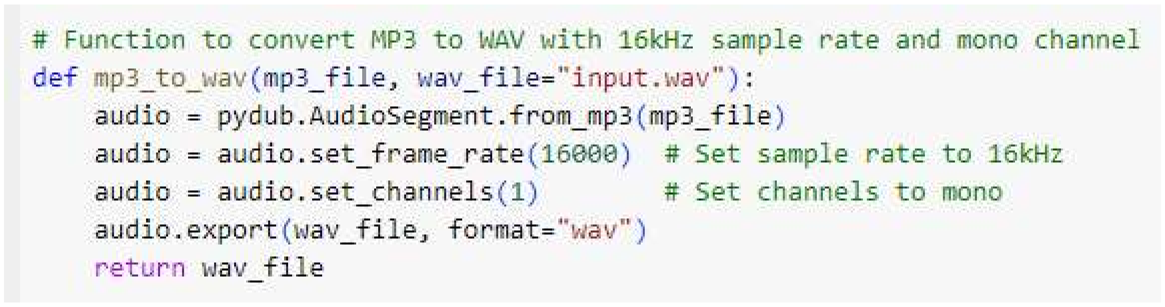


Code Snippets (Python)

1. Audio Preprocessing

MP3 to WAV Conversion

- **Objective**: Convert an MP3 file into a WAV file that meets the requirements for the VAD (Voice Activity Detection) process.
- **Process**:
    - **Loading the MP3 File**: Using the pydub.AudioSegment.from_mp3(mp3_file) function, the script loads the MP3 file into an audio segment object.
    - **Setting Frame Rate**: The audio's frame rate is adjusted to 16kHz (16,000 samples per second) with audio.set_frame_rate(16000). This is crucial because many speech processing tools, including webrtcvad, require a 16kHz sample rate.
    - **Setting Channels to Mono**: The audio is converted to a single channel (mono) using audio.set_channels(1). Most VAD algorithms, including webrtcvad, are designed to work with mono audio.
    - **Exporting to WAV**: Finally, the processed audio is exported as a WAV file using audio.export(wav_file, format="wav"). The WAV format is uncompressed and retains the full quality of the audio, making it suitable for further processing.



```
# Function to convert MP3 to WAV with 16kHz sample rate and mono channel
def mp3_to_wav(mp3_file, wav_file="input.wav"):
    audio = pydub.AudioSegment.from_mp3(mp3_file)
    audio = audio.set_frame_rate(16000)  # Set sample rate to 16kHz
    audio = audio.set_channels(1)        # Set channels to mono
    audio.export(wav_file, format="wav")
    return wav_file
```

2. Voice Activity Detection (VAD)

Purpose and Functionality

- **Objective**: The goal of VAD is to identify and isolate segments of the audio that contain speech, while filtering out silence, background noise, or other non-speech elements.
- **Process**:
    - **Initialization**: The webrtcvad.Vad(aggressiveness) function initializes the VAD with an aggressiveness level ranging from 0 to 3. A higher aggressiveness level makes the VAD

more likely to classify borderline segments as speech, which is useful in noisy environments.

- **Reading WAV File**: The WAV file is opened and read using the wave module. The sample rate, number of channels, and sample width are extracted to ensure they match the expected values (16kHz, mono).
- **Frame Splitting**: The audio data is split into frames, typically of 30ms duration, using int(sample_rate * frame_duration / 1000 * sample_width). Each frame is processed individually by the VAD to determine if it contains speech.
- **Speech Detection**: The vad.is_speech(frame, sample_rate) function processes each frame. If the frame is classified as speech, it is written to the output WAV file. This creates a new WAV file containing only the speech segments, effectively removing silence and background noise.

```
# Function to perform VAD on the audio and save the output
def apply_vad(audio_file, output_file="vad_output.wav",
aggressiveness=3):
    vad = webrtcvad.Vad(aggressiveness)

    with wave.open(audio_file, 'rb') as wf:
        sample_rate = wf.getframerate()
        channels = wf.getnchannels()
        sample_width = wf.getsampwidth()
        assert sample_rate == 16000, "VAD requires 16kHz audio"
        assert channels == 1, "VAD requires mono audio"

        frames = wf.readframes(wf.getnframes())
        # Convert the frames to bytes-like object
        frames = bytearray(frames)

    # Create an output wave file with the same parameters
    with wave.open(output_file, 'wb') as out_wf:
        out_wf.setnchannels(1)
        out_wf.setsampwidth(sample_width)
        out_wf.setframerate(sample_rate)

        # Process frames with VAD
        frame_duration = 30  # ms
        frame_size = int(sample_rate * frame_duration / 1000 *
sample_width)
        num_frames = len(frames) // frame_size
        print(f"Processing {num_frames} frames...")
        for i in range(0, len(frames), frame_size):
            frame = frames[i:i + frame_size]
            if len(frame) < frame_size:
                # Pad the last frame if it's smaller than the required
size
                frame = frame + bytearray(frame_size - len(frame))
            try:
                is_speech = vad.is_speech(bytes(frame), sample_rate)
                if is_speech:
                    out_wf.writeframes(frame)
            except Exception as e:
                print(f"Error processing frame: {e}")

    return output_file
```

3. Speech-to-Text Transcription

Using Whisper Model

- **Objective**: Convert the processed speech segments into text.
- **Process**:
    - **Model Loading**: The Whisper model is loaded using whisper.load_model(model_name). Whisper is a neural network-based automatic speech recognition (ASR) model known for its high accuracy in transcribing spoken language into text.
    - **Transcription**: The VAD-processed WAV file is fed into the model using model.transcribe(audio_file). Whisper performs end-to-end processing, converting the audio directly into a text string. The result is extracted from the returned dictionary and typically includes the recognized text, along with metadata like timestamps and confidence scores.

```
# Function to transcribe audio using Whisper
def transcribe_audio(audio_file, model_name="base"):
    model = whisper.load_model(model_name)
    result = model.transcribe(audio_file)
    return result["text"]
```

4.Text Generation Using Phi-2

Loading and Utilizing Phi-2 Model

- **Objective:** Generate a contextual and meaningful text response based on the transcribed text.
- **Process:**

- **Model and Tokenizer Loading:** The Phi-2 model and its associated tokenizer are loaded using the transformers library. The model is specifically designed for causal language modeling, making it suitable for generating text that follows a given prompt.
- **Pipeline Setup:** A text generation pipeline is created using transformers.pipeline("text-generation", model=model, tokenizer=tokenizer, device_map="auto"). This pipeline seamlessly integrates the model and tokenizer, enabling the generation of coherent text.
- **Text Generation:** The transcribed text is used as the input prompt for the model. The pipeline(prompt, max_length=max_length, ...) function generates the response text. The max_length parameter controls the maximum number of tokens in the output, and num_return_sequences specifies how many different outputs to generate. To ensure the response is concise, the text is split into sentences, and the first max_sentences (typically 2) sentences are retained.

```
# Function to generate text using the  model with a restriction on
output length
def generate_text(prompt, model, tokenizer, max_length=200,
num_return_sequences=1, max_sentences=2):
    pipeline = transformers.pipeline(
        "text-generation",
        model=model,
        tokenizer=tokenizer,
        device_map="auto"
    )
    response = pipeline(
        prompt,
        max_length=max_length,
       num_return_sequences=num_return_sequences,
        truncation=True,
        pad_token_id=tokenizer.eos_token_id
    )[0]['generated_text']

  # Restrict the response to the specified number of sentences
    sentences = response.split('. ')
    if len(sentences) > max_sentences:
        response = '. '.join(sentences[:max_sentences]) + '.'

    return response
```

5. Text-to-Speech Conversion

Using Edge-TTS

- **Objective**: Convert the generated text back into spoken words, with options for voice customization.

- **Process**:
    - **Voice Customization**: The user selects a voice type (male or female), speech rate, and pitch. These options are presented as a list of choices, allowing for a personalized output.
    - **Asynchronous TTS**: The text-to-speech conversion is performed asynchronously using asyncio. The edge_tts.Communicate(text=text, voice=voice, rate=rate, pitch=pitch) function is used to generate speech. The TTS system converts the text into speech with the specified voice settings and saves it as an MP3 file.
    - **Saving Output**: The synthesized speech is saved to the specified output file. The asynchronous nature of this step ensures that the program remains responsive and can handle other tasks concurrently.

```
# Function to convert text to speech using Edge-TTS with tunable
parameters
async def text_to_speech(text, output_file, voice="en-US-
JennyNeural", rate="+0%", pitch="+50Hz"):
    tts = edge_tts.Communicate(text=text, voice=voice, rate=rate,
pitch=pitch)
    await tts.save(output_file)
```

Options for Voice Type, Rate, and Pitch

- **Objective**: Provide users with the flexibility to customize the TTS output to their preferences.
- **Process**:
    - **Voice Type**: Users can choose between a male (en-US-GuyNeural) and a female (en-US-JennyNeural) voice. These voices are part of Microsoft's neural TTS service, which offers high-quality, natural-sounding speech.
    - **Speech Rate**: The rate of speech can be adjusted using predefined options, ranging from significantly slower to significantly faster. This allows the TTS output to be tailored for different use cases, such as clarity or urgency.
    - **Pitch**: Similar to the speech rate, the pitch can be modified to make the voice sound higher or lower. This feature is particularly useful for creating distinct voice profiles or adapting the TTS output to different contexts.

```
# Function to select voice type (Male/Female)
def select_voice():
    print("Select Voice:")
```

```
       options = {
           "1": "en-US-JennyNeural",  # Female Voice
           "2": "en-US-GuyNeural"     # Male Voice
       }
       for key, value in options.items():
           print(f"{key}: {value}")
       choice = input("Enter the number corresponding to your choice:
   ")
       return options.get(choice, "en-US-JennyNeural")

   # Function to select voice rate
   def select_voice_rate():
       print("Select Voice Rate:")
    options = {
           "1": "+0%",     # Normal Rate
           "2": "+10%",    # Slightly Faster
           "3": "+20%",
           "4": "+30%",    # Moderately Faster
           "5": "+50%",
           "6": "+70%",    # Significantly Faster
        "7": "+100%",
           "8": "-10%",    # Slightly Slower
           "9": "-20%",
           "10": "-30%",   # Moderately Slower
           "11": "-50%",
        "12": "-70%",   # Significantly Slower
        "13": "-100%"
    }
    for key, value in options.items():
           print(f"{key}: {value}")
       choice = input("Enter the number corresponding to your choice:
   ")
    return options.get(choice, "+0%")

   # Function to select voice pitch
def select_voice_pitch():
    print("Select Voice Pitch:")
    options = {
           "1": "+0Hz",    # Normal Pitch
        "2": "+50Hz",   # Slightly Higher
           "3": "+100Hz",
        "4": "+200Hz", # Moderately Higher
           "5": "+300Hz",
           "6": "+400Hz", # Significantly Higher
           "7": "+500Hz",
        "8": "-50Hz",   # Slightly Lower
            "9": "-100Hz",
        "10": "-200Hz",# Moderately Lower
            "11": "-300Hz",
            "12": "-400Hz",# Significantly Lower
            "13": "-500Hz"
    }
        for key, value in options.items():
         print(f"{key}: {value}")
        choice = input("Enter the number corresponding to your choice: 
")
        return options.get(choice, "+0Hz")
```

7. Cleanup and File Management

Ensuring Efficient Resource Management

- **Objective**: Remove temporary files created during the processing stages to free up disk space and maintain a clean environment.
- **Process**:
  - **File Deletion**: After the TTS process is complete, the script checks for the existence of temporary files (e.g., the intermediate WAV files) and deletes them. This is done using the os.remove(file_path) function.
  - **Efficiency**: Cleaning up temporary files ensures that the system remains efficient and avoids unnecessary clutter, which could otherwise lead to issues like running out of disk space or encountering file conflicts in subsequent runs.

```
# Clean up the temporary files
    if os.path.exists(vad_output_file):
        os.remove(vad_output_file)
    if os.path.exists(audio_file):
        os.remove(audio_file)
```

8. Main Function Workflow

Bringing It All Together

- **Objective**: Orchestrate the entire pipeline, from input to final output, in a seamless and user-friendly manner.

- **Process**:
    - **Input Handling**: The main function starts by receiving the MP3 file as input. It converts the MP3 to a WAV file suitable for processing.
    - **VAD Processing**: The WAV file is passed through the VAD to isolate speech segments.
    - **Transcription**: The speech segments are transcribed into text using the Whisper model.
    - **Text Generation**: The transcribed text is used to generate a response via the LLaMA model.
    - **Voice Customization**: The user selects the desired voice settings for the TTS process.
    - **Text-to-Speech**: The generated text is converted to speech and saved as an MP3 file.
    - **Cleanup**: Temporary files are deleted, and the final output is ready for use.

```python
# Main function to run the entire pipeline
def main(mp3_file, model_id="microsoft/phi-2", token='',
output_file="output.mp3"):
    # Convert MP3 to WAV
    print("Converting MP3 to WAV...")
    audio_file = mp3_to_wav(mp3_file)

    # Step 1: Apply VAD to the input audio
    print("Applying VAD...")
    vad_output_file = apply_vad(audio_file)

    # Step 2: Transcribe audio to text
    print("Transcribing audio...")
    transcript = transcribe_audio(vad_output_file)
    print("Transcript:", transcript)

    # Step 3: Load the  model and tokenizer
    print("Loading  model...")
    model, tokenizer = load_llama_model(model_id, token)

    # Step 4: Generate a response based on the transcribed text
    print("Generating response...")
    generated_text = generate_text(transcript, model, tokenizer)
    print("Generated Text:", generated_text)

    # Step 5: Select voice type, rate, and pitch
    voice = select_voice()
    rate = select_voice_rate()
    pitch = select_voice_pitch()
    # Step 6: Convert the generated text to speech
    print("Converting text to speech...")
    # Use asyncio.create_task instead of asyncio.run
    asyncio.create_task(text_to_speech(generated_text, output_file,
voice=voice, rate=rate, pitch=pitch))
```

Summary

This end-to-end pipeline provides a comprehensive solution for converting spoken input into a conversational output, with a strong emphasis on user customization and system efficiency. By integrating advanced AI models like Whisper and LLaMA with practical tools like VAD and Edge-TTS, the system achieves high accuracy, flexibility, and performance in a wide range of real-world scenarios.